Atomic Object Factory: Efficient, Consistent and Dependable Support for Distributed Objects

Pierre Sutra, Etienne Rivière, Cristian Cotes, Marc Sánchez Artigas, Pedro Garcia Lopez, Emmanuel Bernard, William Burns, Galder Zamarreño

Atomic Object Factory: Efficient, Consistent and Dependable Support for Distributed Objects

Research report

Pierre Sutra
Télécom SudParis, CNRS
Université Paris-Saclay,
France

Etienne Rivière
University of Neuchâtel,
Switzerland

Cristian Cotes,
Marc Sánchez Artigas,
Pedro Garcia Lopez
Universitat Rovira i Virgili
Tarragona, Spain

Emmanuel Bernard,
William Burns,
Galder Zamarreño
Red Hat

ABSTRACT

This paper introduces Atomic Object Factory (AOF), a novel implementation of the *distributed object* paradigm. AOF allows client applications to transparently manipulate and share remotely-stored objects, while providing strong guarantees on dependability, consistency and persistence. It leverages the support of a listenable key-value store (LKVS), a novel NoSQL storage abstraction that we introduce in this paper. AOF offers integrated language support at the application-facing interface tier. Its middle tier implements the distributed object functionalities atop the LKVS, including coordinated replication. AOF does not rely on a mapping phase and deals with objects end-to-end. It allows the composition of these objects and provide efficient access to their methods. In particular, AOF guarantees disjoint access parallelism: operations on distinct distribued objects make use of distinct LKVS components and execute in parallel. We assess the benefits of our approach in practice by comparing the implementation of a personal cloud storage service using AOF, to an implementation of the same application with a state-of-the-art object-relational mapping over a sharded PostgreSQL database. Our results show that AOF offers a simpler programming framework both in terms of learning time and lines of code, while performing better on average and being more scalable.

Keywords

Distributed Objects, Language support, Dependability, Listenable Key-Value Store, Composability.

1. INTRODUCTION

Context. Programming a highly-available large-scale application in a distributed Cloud environment is a complex task. In particular, proper management of the application state is critical. To this end, developers require strong guarantees on dependability and performance from the data management solutions they employ. They require to be able to manage and persist the application state using a well-defined and simple interface, ideally allowing for complex operations over structured data. These expectations extend to consistency guarantees in presence of concurrent accesses. As importantly, developers also expect the data management solution to be scalable, and to enable the pay-per-use model of Cloud computing through elastic scaling capabilities.

On the one hand, relational databases remain in the majority of cases the solution of choice for structured data management. They support a well-principled and standardized interface for data specification, through the notion of relations and the SQL language. They also offer strong consistency guarantees with the use of transactions supporting the ACID semantics. On the other hand, the most popular programming model for building complex distributed applications is arguably the object-oriented paradigm (OOP). Developers are generally well-trained in languages using this paradigm, and master the associated concepts such as encapsulation, composition, inheritance, delegation, or object-centric synchronization.

A mapping phase is necessary in order to use a database for persisting and sharing data between the components of an object-oriented application. For relational databases, this is the role of an object-relational mapping (ORM). An ORM transforms the application data from the object-oriented in-memory model to the relational model stored remotely, and vice-versa. Objects mapped to the relational model are persisted in the database and updated using transactions. The mapping can be done manually, or automated thanks to various technologies available in different languages, e.g., Hibernate ORM [8] for Java.

Motivation. The use of ORM solutions and relational databases is unfortunately unable to meet all the needs of modern, large-scale object-oriented applications running in the Cloud. Some of these limitations come from the relational database itself, and some come from the the use of a mapping solution between the application and the storage layer. In what follows, we overview this situation.

With relational databases, scalability is achieved through the sharding of relations across multiple semi-independent servers. However, this operation is complex, requires non-trivial changes at the application level, and breaks consistency for transactions that operate across shards. Moreover, re-sharding is a stop-the-world operation that can induce long application downtimes [19]. This generally prevents relational databases from attaining elastic scalability.

A response to the scalability and elasticity defects of relational databases came in the form of NoSQL databases. By offering a simpler data interface, e.g., in the form of a key/value store, and employing placement techniques such as consistent hashing [38], the design of these novel-generation databases focuses on elastic scalability, that is the ability to add/remove servers with limited amount of data migration

and without service interruption. NoSQL databases provide a large variety of consistency guarantees, ranging from eventual consistency [20, 60] to stronger guarantees such as linearizability [16, 23, 49]. Some NoSQL databases also provide transactions with various levels of isolation, e.g., serializability (MarkLogic), snapshot isolation (RavenDB) or repeatable read (Infinispan [49]). Similarly to relational databases, it is possible to define a mapping between in-memory objects and their representations stored in the NoSQL database, either manually or using tools such as Hibernate OGM.

Object-to-database mappings have some intrinsic drawbacks that are commonly named the *impedance mismatch*. These limitations are shared by mapping solutions for both relational and NoSQL databases.

First, the programmer should go back-and-forth between the in-memory representation of the data and its storage representation. This often requires to retrieve and instantiate objects from the persistent storage, access and modify them locally, then eventually push them back. This operation is done at the level of the full object state, which is shared anew upon a mutation. As a consequence, when the objects get large, performance degrades rapidly.

The second problem that arises with ORM is that the relational model is data agnostic and understands solely tuples. This implies, for instance, that constructing even a shared counter in SQL requires some ad-hoc code. The same applies to the NoSQL stores that solely understand opaque values stored under a certain key.

A third problem occurs with the management of complex hierarchical data structures and data locality. Circumventing issues that arise in such situations usually rely on complicated stored procedures. These procedures increases the overall code complexity, while reducing the maintainability and portability of the application.

Contributions. In this paper, we propose that distributed application developers benefit from the scalability, elasticity and dependability of NoSQL systems, while at the same time manipulating the familiar object-oriented programming (OOP) paradigm. We remove the impedance mismatch associated with mapping approaches by avoiding an explicit conversion layer, and by dealing with objects end-to-end from the application code to the storage layer.

Our novel framework, *Atomic Object Factory* (AOF), offers language-level support for data persistence, distribution and concurrency. It follows the *distributed object* paradigm. In comparison to previous attempts implementing this paradigm, e.g., dedicated object-oriented databases or object broker middleware solutions, AOF shows key differences that allow reaching its goals of performance, dependability and usability. First, we do not build a new and specific data management layer but leverage the robustness, elasticity and scalability of a NoSQL data store. Second, AOF offers high availability in the presence of crash failures through the use of state-machine replication. Third, objects can be composed in arbitrary acyclic graphs, and cached close to the client for performance. Finally, the use of AOF is transparent and simple: one just needs to annotate the classes of objects to access them remotely and distribute the computation. Our framework takes complete care of moving the objects and their constituents to the underlying NoSQL data store, replicating them, and providing strong consistency in presence of concurrent accesses.

This paper makes the following contributions:

- The design and implementation of AOF, which is formed of three complementary tiers:
  - ▷ A first application-facing *interface* tier implementing simple and transparent support for the Java language based on annotations, allowing developers to easily manipulate remotely-stored objects;
  - ▷ A second intermediate *object management* tier implementing support for dependable and strongly consistent shared objects;
  - ▷ A third and lower *storage* tier in the form of a listenable key-value store (LKVS), a novel NoSQL abstraction that allows AOF to meet its consistency, dependability and performance guarantees. We provide both an algorithmic realization of the LKVS, and a concrete implementation in a state-of-the-art NoSQL database;

- The description of a representative cloud application in the form of StackSync, a personal cloud storage application. This application initially uses a state-of-the-art ORM solution with sharding and stored procedures. We describe its porting to AOF, and report on the relative complexity of the two approaches;

- An evaluation of the performance and scalability of AOF both using micro-benchmarks and the StackSync application under a large-scale trace from the Ubuntu One personal cloud service. This evaluation shows that AOF consistently outperforms the ORM-based approach, being able to sustain a higher throughput and to offer shorter response times.

Outline. We organize the remaining of this paper as follows. Section 2 presents the interface tier of AOF and its guarantees. Section 3 details the two server-side object management and storage tiers of AOF, and Section 4 covers implementation details. In Section 5, we describe the use case application. Results of our evaluation appear in Section 6. We survey related work in Section 7, then conclude in Section 8. For readability purposes, we defer some algorithmic details as well as our correctness proofs to the appendix.

2. PROGRAMMER'S PERSPECTIVE

AOF simplifies the use of persistent, dependable and scalable storage, when developing a novel application but also when porting some existing code. The programmer accesses a familiar object-oriented interface that allows creating distributed objects and manipulating them with live and strongly consistent operations. This section details the interface AOF offers to the programmer as well as the guarantees she can expect from the framework. These guarantees are enforced by the mechanisms we cover in Section 3. We also delineate the requirements AOF expects for the distributed objects class definitions.

2.1 Interface

We explain how to employ AOF with an example taken from our use case application, StackSync [46]. This personal cloud storage service implements a collection of remote file systems, or *workspaces*, which can be shared between multiple users, similarly as Dropbox. Figure 1 presents two code

samples of the Stacksync application, a `Workspace` and a `Facade` class.

Each `Workspace` object defines an identifier, a root, as well as the collection of users that are allowed to access it. We observe in the declaration of the `Workspace` class the use of the `@Distributed` annotation. This annotation informs AOF that `Workspace` objects should be distributed. The value of parameter `key` indicates that the field `id` uniquely identifies each of the `Workspace` objects.

The `Facade` class is the main entry point for the StackSync application. It offers basic end-user operations, such as the addition of a novel device (see lines 17-19 in Figure 1-bottom). The declaration of this class includes three collections: `devices`, `workspaces` and `users`. AOF instantiates each collection remotely, thanks to the annotation `@Distribute`. The value of `key` (lines 4, 7 and 10) defines the name of the collection.

A field annotated with `@Distribute` is a unique instance accessible from all the client application processes. In other words, each `@Distribute` collection is transparently shared between all `Facade` objects. As a consequence, AOF enforces such fields to be class members, i.e., AOF requires the presence of the `static` keyword (see lines 5, 8 and 11).

When the StackSync application instantiates an object that is declared as distributed, AOF returns a proxy that handles the interaction with the storage tier transparently. In particular, the proxy takes care of retrieving (or creating) the corresponding remote state. This state is updated/read appropriately every time a thread executes a method call, similarly to what happens in Java RMI or CORBA.

2.2 Guarantees

AOF offers the following key functional guarantees to the programmer of a distributed application.

(Strong Consistency) Objects distributed with AOF are linearizable [33] at the interface tier. This means that if two processes operating on any two machines of the interface tier concurrently execute method calls, these calls will appear as executed in some sequential order. Furthermore, this sequential order respects the real-time ordering of calls.

(Composition) AOF allows the programmer to compose objects in arbitrary acyclic graphs. The composition of objects in AOF maintains linearizability. This property is similar to what is offered by a concurrent programming library on a single machine.

(Persistence) Every object created and hosted by AOF is persistent: if the application stops then later restarts, the objects it was manipulating are intact. Such a property allows to easily program stateless application components, improving scalability and dependability.

In addition, our framework can adapt to a workload and leverage its intrinsic parallelism with the help of the non-functional guarantees that follow.

(Disjoint-access Parallelism) Operations that access distinct distributed objects operate on disjoint components of the storage tier.

(Elasticity) AOF can provision/remove servers at the storage tier without interruption.

```
1  @Distributed(key = "id")
2  public class Workspace {
3
4    public UUID id;
5    private Item root;
6    private List<User> users;
7
8  /* ... */
9
10   public boolean isAllowed(User user) {
11     return users.contains(user.getId());
12   }
13 }
```

```
1  @Distributed(key = "id")
2  public class Facade {
3
4    @Distribute(key = "deviceIndex")
5    public static Map<UUID,Device> devices;
6
7    @Distribute(key = "workspaceIndex")
8    public static Map<UUID,Workspace> workspaces;
9
10   @Distribute(key = "userIndex")
11   public static Map<UUID,User> users;
12
13   public UUID id;
14
15 /* ... */
16
17   public boolean add(Device device) {
18     return deviceMap.putIfAbsent(
19       device.getId(),device) == null;
20   }
21 }
```

Figure 1: Workspace and Facade classes

2.3 Definition and Requirements

We formalize the definition of distributed objects as follows. Consider some object o of type T. We say that o is a *distributed object* when either o is a static field annotated with `@Distribute`$(key = v)$, or the type T is annotated with `@Distributed`$(key = f)$. Let us note k the value of v in the former case, and the value of the field f in the latter. The pair (T, k) is the *reference* to object o, denoted hereafter ref_o. This reference uniquely identifies o in the system.

AOF puts two requirements on the type T of a distributed object. The first requirement is linked to the use of state machine replication to ensure dependability. It states that T is a deterministic sequential data type. Such a type enforces that from a given state the object reaches a unique new state and returns a single response value. Formally, T is a tuple $(States, S_0, Methods, Values, \tau)$ where $States$ is the set of states, $S_0 \subseteq States$ are the initial states, $Methods$ defines the methods, $Values$ are the response values, and $\tau : States \times Methods \rightarrow States \times Values$ is the transition function. Hereafter, and without lack of generality, we shall assume that $Methods$ is *total*, meaning that for every method m, $States \times \{m\}$ belongs to the domain of τ. Our second requirement is that the type T is *marshallable*, i.e., any object of type T can be transferred over the wire. The process of marshalling must also be deterministic.

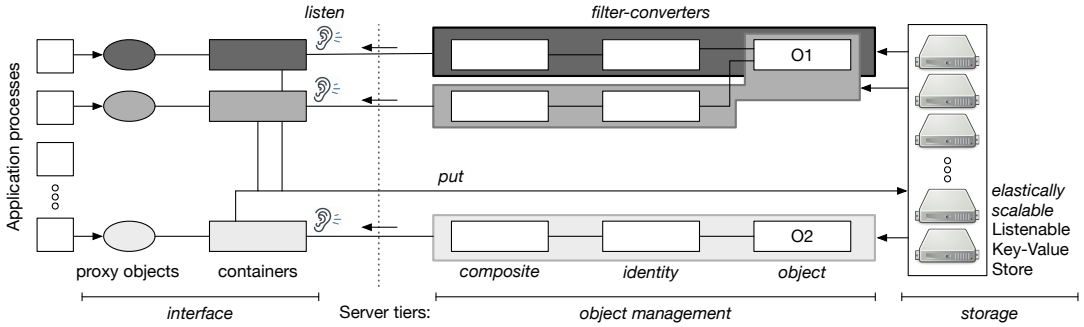


Figure 2: AOF architecture

3. AOF DESIGN

Figure 2 presents the general architecture of AOF. Our framework is composed of three tiers: *interface*, *object management* and *storage*. In what follows, we provide first an overview of each tier then detail the lifetime of an object in AOF. Further, we explain the composition of objects and the client-side caching mechanism. Our design ensures that AOF maintains all the distributed objects accessed at the interface tier linearizable, live and persistent. A detailed proof of correctness appears in Appendix B.

3.1 Overview

The lowest tier of AOF is the *storage* tier. We define for this tier a novel NoSQL storage abstraction, the *listenable key-value store* (LKVS). In addition to the usual *put*() and *get*() operations, a LKVS allows registering an external listener on a particular key, together with an associated filter-converter. In a nutshell, when a *put*() operation occurs on the key, the LKVS calls the filter-converter to transform the new value before sending it to the listener.

The interface tier consists of a set of containers. A container holds the proxy that the application accesses in lieu of the object itself [54]. This proxy is created when the application instantiates an object annotated `@Distributed` or `@Distribute`. The first method call registers the container as a listener of the LKVS with an associated filter-converter. For instance, in Figure 2, the light gray container is linked to the light gray filter-converter.

The filter-converters define the object management tier. They support a collection of elastic, durable and consistent shared objects stored in the LKVS. This tier implements an extension of the seminal state machine replication approach [53], which is able to deal with the lifetimes of objects and their composition.

Each filter-converter is formed of three components, named *composite*, *identity* and *object*. The composite and identity components are unique for a given filter-converter and thus for a unique application-side container. They coordinate the accesses by one given container. The object component coordinates the accesses to the actual objects state and lifetime. In the LKVS, at most one object component exists per distributed object created by the application. This component is therefore shared by several filter-converters if different processes of the application have opened the same distributed object. For instance, in Figure 2, the two filter-converters with the darkest shades of gray share the same object component O1.

At the interface side, the container translates every call to some method m on a distributed object into a *put*() operation on the LKVS. This operation triggers a computation in the filter-converter(s) that coordinate the execution of the method. Since the calling container is also registered as a listener on the LKVS, it eventually receives a notification that this method call occurred, together with the result.

3.2 Listenable Key-Value Store

We now detail the interface and guarantees the listenable key-value store (LKVS) offers.

Interface. A LKVS exposes a classical map interface to *put*() and *get*() values. In addition, it allows pluggable pieces of code to listen to its modifications, in the form of a *listener* together with an associated *filter-converter*. With more details, a listenable key-value store presents the following operations at its interface:

- $put(k, u)$ stores value u at key k;
- $get(k)$ returns the value stored at key k, or $\perp$ if no such value exists;
- $regListener(k, l, f_l)$ registers the listener l and its associated filter-converter f_l at key k;
- $unregListener(k, l)$ unregisters the listener l, removing both l and f_l from the LKVS.

A listener receives the events sent by its associated filter-converter. While the former is typically a distant process, the latter is executed at the LKVS side. More precisely, registering a listener l together with its filter-converter f_l at some key k has the following semantics: Consider that a call to $put(k, u)$ occurs, and note v the old value stored under key k. A filter-converter defines a filtering function $filter(u, v)$. If $filter(u, v)$ returns $w \neq \perp$, then listener l eventually receives at least once the event $\langle k, w \rangle$. In such a case, we shall say that the event $\langle k, w \rangle$ *passes* the filter-converter f_l.

Guarantees. The LKVS offers guarantees that are the basis of the guarantees AOF provides. As defined in Section 2.2, we are interested in this paper with providing strong consistency for concurrent accesses. We also aim at ensuring the dependability of distributed objects through the use of state machine replication. These two goals require the LKVS interface to be *wait-free* as well as *linearizable*. Wait-freedom ensures that any operation issued by a LKVS client eventually returns. Linearizability satisfies that *(i)* every entry of the LKVS map behaves like an atomic register, and *(ii)* once a listener l is registered, l receives all the events $\langle k, filter(u, v) \rangle$ that pass the filter-converter f_l in the order

their associated operations $put(k, u)$ are linearized.

In addition to the above guarantees, we consider that every filter-converter associated with a listener is kept into the LKVS until that listener is unregistered, or until the (client) process on which l resides fails. As we will make clear later, this last mechanism is here for the purpose of bounding memory usage and does not harm dependability.

In Section 2.2, we list disjoint-access parallelism and elasticity as two key non-functional guarantees AOF offers. Appendix A details how we can meet these guarantees through a proper implementation of the LKVS.

3.3 Object Lifetime

When a client application invokes the constructor of a distributed object o of type T with arguments $args$, AOF calls instead $newObject(k, T, args)$ to create object o remotely. This operation instantiates a *container* c that holds the reference of the object (with $ref_o = (T, k)$), as well as the proxy object returned in place of o to the application.

Creation and Disposal. The first call to some method of the proxy object *opens* container c. When it is opened, container c registers itself as a listener to the LKVS with an associated filter-converter f_c. This filter-converter takes a key part in the object management tier on the LKVS servers. It holds the state of the object, and coordinates accesses to its methods with the other filter-converters registered as listeners under key ref_o.

Once container c is opened, a call to some method m of the proxy object *(i)* registers a future [6] for that call, *(ii)* executes $put(ref_o, \langle \texttt{INV}, c, m \rangle)$ on the LKVS to invoke the method on the distributed object, then *(iii)* waits until the future completes. The filter-converter corresponding to the distributed object o executes the method m and notifies the future created by the listener with the result.

At some application process, AOF tracks a bounded amount of containers. To this end, we maintain a cache of containers, and apply an LRU eviction policy to the cache. Once a container is evicted from the cache, AOF closes it. Closing the container c of some distributed object o consists in executing $\mathcal{K}.put(ref_o, \langle \texttt{CLOSE}, c \rangle)$ then unregistering c as a LKVS listener. Every subsequent call to the proxy object will re-create an appropriate container.

Filter-Converters. AOF manages the distributed objects created at its interface inside the filter-converters. As illustrated in Figure 2, each filter-converter consists of three components. These components implement the same logic as the filter-converter itself, namely a filtering function $filter(u, v)$ that returns $\perp$ when the event $\langle k, u \rangle$ does not pass the filter-converter. We detail the roles of these components in what follows.

The *composite component* defines the entry and exit points of the filter-converter. Internally, it calls the two other components: first the identity component is called, and if the event passes through, the result is then injected into the object component. The result of this last call is the return value of the filter-converter.

Consider that some container c installs a filter-converter. The *identity component* determines when c needs to be notified of the execution of a method on the distributed object. Indeed, as the object component is shared between several filter-converters, not only the filter-converter of c is called but also those of other containers. For every novel insertion $put(k, u)$ in the LKVS, the identity component returns u in case c is mentioned. In practice, this translates into the fact that c appears in the parameters of the insertion, e.g., $put(ref_o, \langle \texttt{INV}, c, m \rangle)$. If c is not mentioned, the identity component returns $\perp$.

Algorithm 1 Object Component of a Filter-Converter

```
1:  Variables:
2:      K                                              // LKVS
3:      s                                              // Initially, ⊥
4:      openContainers                                 // Initially, ∅
5:
6:  filter(u, v) :=
7:
8:          pre:  u = ⟨OPEN, c, args⟩
9:                s ≠ ⊥ ∨ v = ⊥ ∨ v = ⟨PER, _, ŝ⟩
10:         eff:  openContainers ← openContainers ∪ {c}
11:               if s = ⊥
12:                 if v = ⊥
13:                   s ← new T(args)
14:                 else
15:                   s ← ŝ
16:               return ⟨ACK⟩
17:
18:         pre:  u = ⟨INV, c, m⟩
19:               s ≠ ⊥
20:         eff:  (s, r) ← τ(s, m)
21:               return ⟨ACK, r⟩
22:
23:         pre:  u = ⟨CLOSE, c⟩
24:         eff:  openContainers ← openContainers \ {c}
25:               if |openContainers| = 0
26:                 K.put(ref_o, ⟨PER, c, s⟩)
27:               return ⟨ACK⟩
28:
29:         pre:  u = ⟨PER, _, _, ⟩
30:         eff:  if |openContainers| = 0
31:                 s ← ⊥
32:               return ⊥
```

The *object component* maintains the actual state of the distributed object on the LKVS. It is in charge of executing the method calls to the object. For every distributed object o, a single object component exists. It is shared between all filter-converters registered under key ref_o. This component is initially created by the first container that opens o.

Algorithm 1 describes the internals of the object component of a filter-converter. It consists in a sequence of effects (**eff**), each guarded by one or more preconditions (**pre**). When all the conditions in one of the **pre** clauses are true, the instructions in the corresponding **eff** block are triggered. Note that all the **pre** conditions are mutually exclusive in Algorithm 1, henceforth their order of evaluation does not matter.

For some object o, the object component maintains three variables: the state of the object (variable s), a reference to the LKVS (variable $\mathcal{K}$), and a list of open containers accessing it (variable *openContainers*).

When a modification occurs at key ref_o, the object component receives a pair (u, v), where u is the novel value of k, and v the old one (line 6 in Algorithm 1). Depending on the content of u, the object component executes either an open (line 8), an invocation (line 18), a close (line 23) or a persist (line 29) operation. We detail each of these cases below.

(OPEN) When a client opens a container c, it executes a call $\mathcal{K}.put(ref_o, \langle \texttt{OPEN}, c, args, \rangle)$ at the LKVS interface. Upon

receiving a tuple $\langle$`OPEN`$, c, args, \rangle$ the object component fetches o from persistent storage or creates it if necessary. In this latter case, the object component uses the constructor that takes as input parameters *args* (line 13). According to the precondition at line 9, this computation takes place only if (1) the object exists at that time ($s \neq \bot$), (2) it was not created in the past ($s = \bot$), or (3) the object component can retrieve it from persistent storage ($v = \langle$`PER`$, \_, \hat{s}\rangle$). In the second case, the object component initializes the distributed object (line 13); whereas in the third case, it uses the previous value (line 15). We can prove that, at any point in the execution, one of the above three predicates hold. This implies that every container opening is wait-free.

(`INV`) Upon receiving a method call to the distributed object (in the form of a tuple $\langle$`INV`$, c, m\rangle$), the object component executes it on s and returns the result r (line 21). The composite component will receive this result and trigger a listener notification back to the caller of method m.

(`CLOSE`) When the client application closes locally the container holding the reference to o, it executes a call to $\mathcal{K}.put(ref_o, \langle$`CLOSE`$, c\rangle)$. Upon receiving this call, the object component removes c from the set of open containers (line 24), then sends back an acknowledgment (line 27). Furthermore, if all the containers are closed (line 25), the object component persists the object state using a call to $\mathcal{K}.put(ref_o, \langle$`PER`$, c, s\rangle)$.

(`PER`) Upon receiving a tuple flagged `PER`, the object component forgets the object state if all the containers remains closed (line 31). Then, it returns $\bot$ to skip notifying the listeners.

Replication. Under the hood, each filter-converter is replicated at the LKVS level. Thanks to the linearizability of the LKVS, all replicas of the same filter-converter receive *put*() operations in the same order. As the distributed object type is required to be deterministic (see Section 2.3), replicas of the same object component are kept consistent across calls, implementing state machine replication.

3.4 Composability

Figure 1 illustrates that AOF supports the composition of distributed objects. Indeed, these definitions tell us that both the `Facade` and `Workspace` classes are distributed, and that each `Facade` object references a map of `Workspace` objects (via the `workspaces` class field).

To implement composability, AOF allows object components to use containers. This means that when an object component executes the code of a method (line 20 in Algorithm 1), it may call a distributed object. More precisely, when either one of the object's fields, or one of the arguments of the method is distributed, the object component opens the corresponding container and executes a remote call, as a regular application process would do.

Composition is not granted for free. We have to keep in mind two considerations when composing objects.

(Call Acyclicity) A method call executes synchronously at the object component. As a consequence, the application should proscribe the creation of call loops across several objects. We notice here that call cyclicity can sometimes (but not always) be detected at compilation time. Internally, AOF avoids the problem of entering call loops with the `hashcode` and `equals` methods as follows: Upon a call to `hashcode`, AOF returns the hash code of the reference of the object held by the container. Now in the case of `equals`, if the operand is also a proxy, their references are compared; otherwise, `equals` is called on the operand with the proxy as argument.

(Call Idempotence) AOF ensures that each method call is idempotent. This property comes from the fact that events are received *at least once* at a listener. More precisely, we explained in Section 3.3 that a copy of the object component associated with some object o is kept at all the replicas of key ref_o. Consequently, when an application process calls method `addDevice` on a facade object f (see Figure 1), every replica of ref_f executes a call to `putIfAbsent` on the `devices` index. Without idempotence, a replica of ref_f may return *false*, even if the call actually succeeded in the first place. Section 4 explains in details how AOF implements call idempotence.

3.5 Extensions

AOF supports client-side caching of objects for methods that are read-only. These methods should be annotated by the programmer with the `@Readonly` keyword. If this occurs, when a method m is called, the object component returns not only the response value of m, but also the the full state of the object. This state is stored locally in the container, and every call to a `@Readonly` method evaluates on this locally cached state. Notice that the use of this mechanism comes at the price of weaker consistency, namely sequential consistency [42]: updates to the object are seen in a common sequential order at the interface tier, but this order might not respect real-time ordering.

Execution of method calls on application objects at the storage tier may lead to system-wide performance issues; for instance when encountering an infinite loop. The solution proposed by the authors of [21] is to restrict the loops used by operations to system-provided iterators with termination guarantees. We postpone to future work the addition of a similar mechanism for AOF, and how to address issues related to isolation in general.

4. AOF IMPLEMENTATION

We implemented AOF on top of Infinispan [49], an industrial grade open-source key-value store. The base code of AOF is in Java and accounts for 4,000 SLOC. To implement the full support of the LKVS abstraction, we contributed a batch of modifications to Infinispan that weight around 13,500 SLOC.

In comparison to the implementation of the AOF constituents we described in Section 3, our code also includes a few additional practical features. We detail such features next, as well as the interesting aspects of our implementation.

Language Support. We added the Java support of AOF specific annotations with AspectJ [40]. This support allows a developer to implement her application without distribution in mind. Under the hood, AOF defines two aspects to marshall and distribute objects. When it encounters a call to a constructor of an object declaring an AOF-specific annotation, the AspectJ compiler replaces this call with the

creation of a container (using aspect weaving). This container returns a proxy that traps all the method calls to the object.

Marshalling. When a distributed object is marshalled, the container triggers a `writereplace` method that returns the object's reference. The actual marshalling of the object in AOF (at line 26 in Algorithm 1) is however not weaved. When this computation occurs, AOF needs to access all of the appropriate fields. In the current state of our implementation, we simply mark such fields public. Another possible approach would be to declare appropriate getters, similarly to what exist with JavaBeans.

Data Persistence. The persistence of distributed objects implemented with AOF relies on the underlying LKVS. Namely, once all containers referring to an object are closed, AOF executes a *put*() operation with the current object state as argument (line 26). To improve performance in our implementation, this call is asynchronous and modifies only the local storage of the LKVS nodes in charge of the object component. Similarly to the synchronous case, when the object component receives the tuple $\langle \texttt{PER}, \_, \_ \rangle$, it forgets the object state if no container remains opened (lines 30 to 32).

Listener Mutualization. The cost of installing a listener per distributed object can be high. In particular, on a machine supporting the application, for each distributed object a naive implementation would use one open connection to the storage tier. Instead, the implementation of AOF multiplexes the listeners for the same application process, and thus only opens a few connections in total.

Chaining Calls. AOF allows chaining calls between objects, provided that no cycles are created. Starting from the application process, a call chain traverses multiple object components located on different LKVS machines. In this case, as mentioned in Section 3.4, several machines may execute the same method call (at line 20 in Algorithm 1).

AOF solves this problem by enforcing call idempotence. With more details, each call has a unique ID. Once a method m returns at line 20, the object component stores its return value r. If a call with the same ID is received, the object component simply returns r. The call ID is computed by the container with the help of a (deterministic) generator. When the object filter-converter component calls a method m at line 20, it also defines the generator using a `ThreadLocal` variable. This generator is used to generate all the call IDs to distributed objects used inside method m.

Garbage collection. AOF manages the garbage collection of distributed objects at both the storage and interface tiers. To this end, AOF maintains a cache of open containers. When a container is removed from the cache, AOF closes it. A process wishing to access the corresponding object will have to recreate a new container. Furthermore, at the object filter-converter level, when the object is closed by all the remote containers, AOF persists the state of the object on the LKVS (line 26 in Algorithm 1) and garbage-collect the object filter-converter component. These two mechanisms ensure that the memory usage of the distributed objects mechanism remains bounded.

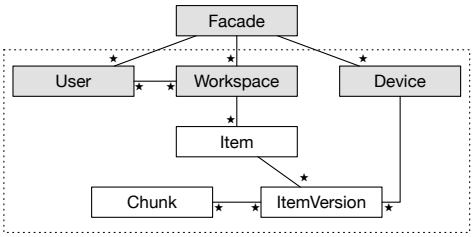


Figure 3: SyncService UML class diagram – (PostgreSQL) dotted border enclose tables, (AOF) grayed classes are `@distributed`

5. USE CASE

In this section, we illustrate the benefits of AOF in a real-world usage scenario. Our target use case is a personal cloud storage service. In what follows, we present this service then cover two implementations: one relying on an object-relational mapping, and another using AOF.

Context. Personal cloud storage services such as Dropbox, Ubuntu One, Google Drive or Box are distributed systems that synchronize files across multiple devices with a back end operating in the Cloud. They typically rely on NoSQL databases for their data, and relational databases to store their metadata. Indeed, these services require strong consistency guarantees to keep their files up-to-date, and relational databases are the de-facto standard for these requirements.

But relational databases also pose some core problems in this context. One important issue is the *expressiveness* of SQL to implement recursive data structures. Such services maintain metadata for the file system of every user, including directories, files, chunks and versions. These structures may be complex to model in SQL to ensure queries efficiency.

Another relevant problem is *scalability*. This is achieved through the use of relational sharding. As we detailed in Section 1, sharding is a difficult operation, and it requires non-trivial changes to the application. It also drops the ACID guarantees for transactions that touch multiple shards, necessitating great care when deciding on the sharding plan. The difficulty of sharding is also related to load balancing. A measurement study on existing personal cloud storage services [22] shows that a small fraction of very active users account for the majority of the traffic, while there is an important population of almost idle users (65% in the *Ubuntu One* service and 30% in Dropbox). Sharding inadequately may lead to serious load imbalance, something difficult to predict before the sharding plan is actually put in use.

A third important issue is regarding *elasticity*. Several past works [22, 28, 45] indicate that existing personal cloud storage services present strong diurnal seasonality. The problem of supporting elasticity with a sharded relational database was already pointed out in our introduction: it requires stopping the service, re-sharding and restarting, which is not a viable option for a production service. In fact, the classical solution in this case is over-provisioning to avoid re-sharding.

Overview. Our use case application is StackSync [46], an open-source personal cloud storage service. StackSync is implemented in Java using an object storage service (OpenStack Swift) for its data, and a relational database (PostgreSQL) for its metadata. The StackSync synchronization service, or *SyncService* hereafter, uses ORM technologies to persist

metadata in the relational database. In what follows, we present the ORM-based implementation and our portage to AOF. These two approaches manipulate the same metadata objects detailed below.

Metadata. The SyncService operates several multi-threaded instances. Each instance processes incoming StackSync requests to modify the application metadata. We depict the class schema of the SyncService in Figure 3. At some SyncService instance, a thread interacts with the storage tier using a `Facade` object. Classes including the `Facade` and below differ from one persistence technology to another. In addition to the configuration and deployment of the storage tier, their portage is where we spent most of our effort.

A `Workspace` object models a synced folder. Every such object is composed of files and directories (`Item` in Figure 3). For each `Item`, the SyncService stores a versioning information in the form of an `ItemVersion`. Similarly to other personal cloud storage services, StackSync operates at a the sub-file level by splitting files into chunks; this greatly reduces the cost of data synchronization. A `Chunk` is immutable, identified with a fingerprint (SHA-256 hash), and it may appear in one or more files.

Relational Approach. To scale up the original relational implementation, we followed conventional wisdom and sharded metadata across multiple servers. The key enabler of this process is PL/Proxy, a stored procedure language for PostgreSQL. PL/Proxy allows dispatching requests to several PostgreSQL servers. It was originally developed by Skype to scale up their services to millions of users.

Following this approach, we horizontally partition metadata by hashing user ids with PostgreSQL built-in function. As a result, all the metadata of a user is slotted into the same shard. Any request for committing changes made by the same user is redirected to the appropriate shard. In detail, we accomplish this with the following PL/Proxy procedure (we omit some parameters for readability):

```
1 CREATE OR REPLACE FUNCTION commitItem(client_id uuid, ...)
2 RETURNS TABLE(item_id bigint, ...) AS $$
3     CLUSTER 'usercluster';
4     RUN ON hashtext(client_id::text);
5 $$ LANGUAGE plproxy;
```

In the above code, the proxy first picks a shard with `RUN ON hashtext(client_id::text)`. Then, it runs a stored procedure named `commitItem` at the chosen shard. This procedure contains the logic for committing a `Workspace` changes in PostgreSQL. It re-implements the original SQL transaction for committing changes to a `Workspace` (`commitRequest` in [46]). At a high level, committing a change requires several SQL insertions to different tables: one `INSERT` statement to create a new version of the file, and a variable number of SQL insertions to the table that records the chunks that constitute the file. We further detail how the stored procedure `commitItem` works in Section 6.2.1.

At the time of this writing, there is no other mature technology to create a virtual table spread across multiple PostgreSQL servers. While the PL/Proxy approach has its pros, e.g., having SQL statements operate close to the data, it also presents several caveats. First, it requires encapsulating all SQL statements in server side functions. This adds significant complexity to the development and maintenance of the application logic. The amount of changes in lines of code (LoC) were of several thousands, and it took us multiple weeks to obtain a stable system. Second, our implementation only scales up plain file synchronization. Partitioning is at odds with sharing. To allow two users to share a folder while maintaining consistent operations, the system would need to support cross-partition transactions. However, PL/Proxy currently does not provide such a feature.

Portage to AOF. AOF fully embraces the OOP paradigm, and this greatly helped us to simplify the code of the SyncService. Starting from the PostgreSQL code, we removed the overhead of coding in SQL, and the use of low-level procedural PL/PostgreSQL constructions. The portage to AOF was split in two steps. First, we moved all the application logic back to the classes, as required by the OOP paradigm. As we benefited from the high-level construct from Java, this phase took only a couple of days. Overall, we wrote fewer lines of code, and in our honest opinion, this code is easier to understand.

In a second step, we annotated the key classes and fields implementing the StackSync back end with `@Distributed` and `@Distribute` (see Figure 3). The key question we had to address here was "What classes do we distribute?". In a basic design, we distributed all the classes. As the performances were lower than those of the PostgreSQL implementation, we then embedded some classes into others. Namely, `Item`, `ItemVersion` and `Chunk` were all embedded in `Workspace`. We ended-up with the set of grayed classes in Figure 3 as being `@Distributed`, and a few static variables in `Facade` holding the global indexes annotated with `@Distribute` (see Figure 1). This configuration has the advantage to maintain operations across workspaces atomic, which would be difficult to achieve using PL/Proxy and sharding. Another option would be to distribute only the `Facade` objects, which would result in non-atomic cross-workspace operations as with PL/Proxy.

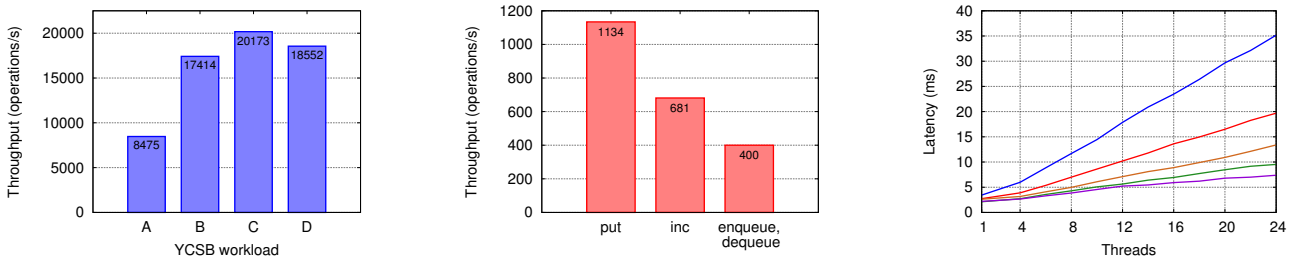
6. EVALUATION

In this section, we evaluate the performance of AOF with micro-benchmarks and under our use case application StackSync.

6.1 Micro-benchmarks

We first consider a typical deployment of AOF in the Cloud based on virtualized commodity servers. With more details, our experimental deployment consists in a cluster of virtualized 8-core Xeon 2.5 Ghz machines each with 8 GB of memory, running Ubuntu Linux 14.04 LTS, and connected by a virtualized 1 Gbps switched network. Network performance, as measured by *ping* and *netperf*, is of 0.3 ms for a round-trip with a bandwidth of 117 MB/s. At each Infinispan server instance, objects resides in a cache. Passivation to persistent storage occurs in the background when 10^5 objects are present in the cache, and it follows an LRU strategy. The (virtual) hard-drive read/write performance is 246/200 MB/s. Unless otherwise stated, the replication factor of any object is set to 2, and we use 3 Infinispan servers.

Raw Performance. We first evaluate the performance of Infinispan, implementing the LKVS that supports AOF. We use to this end the Yahoo! Cloud Serving Benchmark (YCSB) [18]. YCSB is a data storage benchmark that consists



(a) YCSB workloads on Infinispan

(b) Accessing a single key: Infinispan put() and AOF methods calls

(c) Concurrent increments – from top to bottom 1, 2, 5, 10 and 100 counters

Figure 4: Throughput of Infinispan and performance of AOF for single and multiple objects

of four workloads of read/write operations.[1] Each workload is representative of a typical Cloud operation: workload A is update-heavy, B is read-heavy, C is read-only, and workload D consists in repeated reads (95% of operations), followed by insertions of new values. All four workloads access a large set of keys with 10^6 items of 1 KB and should therefore benefit from disjoint-access parallelism.

Figure 4(a) shows the maximal throughput for each of the four different YCSB workloads. We observe that read-dominated workloads achieve around 17 to 20 Kops/sec, while the sole update-heavy workload is capped at around 8.5 Kops/sec. This performance is in-line with the other industrial-grade NoSQL solutions [18].

Simple Distributed Objects. In Figure 4(b), we plot performance over a single key for a *put*() operation on the LKVS, and two operations over simple distributed objects. The first operation uses a shared counter with an increment method (`inc`). The second uses two method calls executed back-to-back by the client that respectively en-queues (`enq`) then de-queues (`deq`) some random elements. The latter two benchmarks are similar to the ones found in [21].

The maximal throughput of a *put*() operation occurring on the same item is $1,134$ ops/sec. This value is an upper bound on the maximal throughput of a method call when accessing an object distributed with AOF. Figure 4(b) also tells us that AOF processes 625 increments per second on a shared counter. When two operations are consecutive, the throughput is as expected almost divided by 2. This is the case for an en-queue followed by a de-queue (column "`enq; deq`" in Figure 4(b)).

When the application increases its parallelism, AOF is able to sustain many more operations per second. We illustrate this in Figure 4(c). This figure depicts the latency of an increment in relation to contention between concurrent application threads in accessing a shared counter. Due to disjoint access parallelism, performance is proportional to the number of shared counters. From top to bottom, we change the number of counters available in the system. With 24 threads and 100 counters, AOF is able to sustain $3,400$ ops/sec; four times the speed at which these threads would access a single counter.

6.2 StackSync

This section evaluates the StackSync service comparatively using PostgreSQL and AOF. To conduct this comparison, we use two different workloads, (i) a synthetic workload consisting of random generated metadata, and (ii) a realistic workload using the Ubuntu One trace [29]. We use the former to evaluate the peak performance of the two systems, and the latter to analyze the behavior of the metadata back end in a realistic scenario.

At core of our two workloads, StackSync users add items to workspaces. The entry point of this functionality is the `doCommit` method of the `Facade` object. This method receives as input a workspace, a user and a device, as well as a list of items to be added to the target workspace. Below, we briefly describe how the `doCommit` method is implemented in PostgreSQL. The AOF implementation is similar following the OOP paradigm.

6.2.1 *Workload Description and Settings*

The `doCommit` method is implemented as a transactional stored procedure in PostgreSQL. This procedure first loads from the database the workspace, the user and the device metadata, and executes base control access and existence tests. Then, it iterates over the list of added items and calls the `commitItem` procedure on each of them.

For each item, the `commitItem` procedure checks if it already exists in the database. In this is not the case, the first version of the item is created and added to the database. Otherwise, `commitItem` produces a new version. In both cases, `commitItem` checks that the added version is appropriate according to the metadata provided by the client (i.e., no concurrent data item creations). If this test fails, the commit is rejected and the sync-service sends the latest version of the faulty item back to the client.

In all the experiments that follow, we fix the number of users of the personal cloud storage service to 4,000. We also assign a unique workspace to each user. A call to `doCommit` consists in the addition of 20 new items to the workspace.

Impact of dependability. In our first experiment, we consider a single StackSync server that accesses the AOF deployment. This server is multi-threaded and makes uses of 240 concurrent threads. Figure 5 reports the results we obtained when data persistence is on (at most 1,000 entries cached per AOF node), and when the replication degree varies between 1 and 2. For each experiment, the StackSync server executes 10^6 operations, and we plot the average. Let us also point out that, in order to produce Figure 5, we simply added AOF nodes at the end of each experiment, relying on the elasticity of our framework.

[1] Infinispan does not support scan operations.

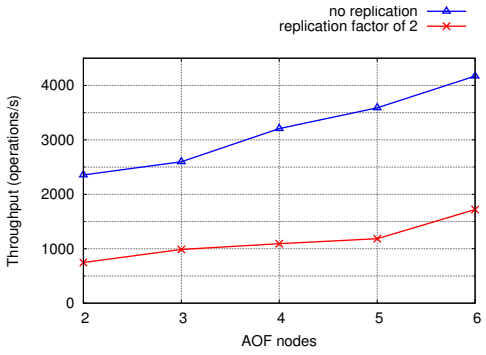


Figure 5: Impact of replication (persistence is on)

As mentioned in Section 5, we opt for distributing all grayed classes in Figure 3, in order to allow atomic cross-workspace operations in the AOF portage. Coordinating more distributed classes is expected to be more expensive than when distributing a single one (e.g., only the `Facade`).

When distributing all grayed classes in Figure 3, and when the replication factor equals two, the performance improvement of AOF equals 1.74 from 2 to 6 servers. This is also very close to the scale-up factor with a replication factor of one (1.77 in that case). Additional results (not presented in Figure 5) indicate that if we only distribute the `Facade` objects and assign each workspace to a single `Facade`, AOF with a replication factor of two sustains 2.7 Kop/sec with 2 nodes, and 5.6 Kop/sec with 6 nodes. This last design offers similar guarantees to the sharding with PL/Proxy, and in particular does not provide the atomicity guarantees on cross-workspace operations that the previous design provides. These results also indicate that if we decrease data persistence, moving the cache size from 10^3 to 10^5 entries, AOF throughput improves by around 30%.

To further understand the performance of AOF and compare it to PostgreSQL, it is necessary to ensure that the StackSync server is not the bottleneck. We solve this problem with the help of a queue messaging service (RabbitMQ). Initially, this service is pre-loaded with either synthetic messages, or messages extracted from the Ubuntu One trace.

Based on the above architecture, the testbed we use for our remaining experiments consists in the following settings: *(i)* Storage: 2 to 6 servers consisting of 4 cores CPU and 8 GB of RAM, and running either PostgreSQL and PL/Proxy or AOF with data persistence on; *(ii)* StackSync: up to 24 single core CPU and 2 GB of RAM running the StackSync service; and *(iii)* RabbitMQ: one server with 6 core CPU and 16 GB of RAM. We should note that in this setting PostgreSQL ensures a higher level of data persistence than AOF. This comes from the fact that each write despite being asynchronous updates the state, while AOF waits for the open containers to close all their references to the object.

Synthetic workload. In this experiment, we pre-load RabbitMQ with 4 Million commits operations. Then, at the same time, we start the 24 StackSync servers, using 10 threads for each of them. This setting corresponds to the peak performance of PostgreSQL in our testbed. StackSync servers process commit operations concurrently at their maximum throughput.

Figure 6(a) depicts the peak performance of PostgreSQL when we use 2 and 4 shards. Notice that in this last case, we deployed 2 proxies; otherwise with 4 shards a single proxy is bottlenecking. The performance of AOF using a single replica per item, and data persistence is depicted in Figure 6(b). On average, AOF is close or better than PostgreSQL in all cases. This comes from the fact that proxies are used solely to forward transactions and not to execute them. With 6 nodes, AOF performance improvement is close to 50%.

Ubuntu One Trace. Figure 6(c) depicts the cumulative distribution of the latency of `doCommit` calls when following the Ubuntu One trace. Based on our previous results, we use 4 shards and 2 proxies for PostgreSQL, while AOF makes use of 6 nodes with a replication factor equals to one and data persistence on.

The trace is executed without a speed-up factor, simply by adding all its operations to the RabbitMQ server. On average, the trace requires 300 `doCommit` operations per second and it lasts several hours. Figure 6(c) tells us that AOF consistently responds quicker than PostgreSQL for this workload.

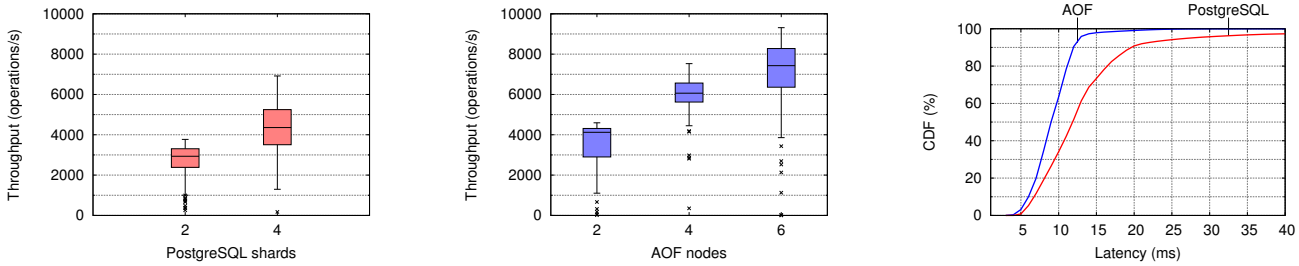
7. RELATED WORK

AOF implements the *distributed object paradigm*, offering language-level support for data persistence, distribution and concurrency. The idea of distributing object takes its roots in remote procedure calls [11], and the ability to load custom procedures in the object store [56]. These approaches hide low-level mechanisms as much as possible, and embrace transparency as a first class principle.

Early solutions. In Argus, Liskov [44] introduces the notion of guardian, a distributed object that exports a set of remote handlers. Calling a handler spawns a thread on a distant machine to execute it. This system also supports (nested) transactions across several guardians. Arjuna [52] is a C++ framework to program distributed application similar in spirit to Argus. Both systems rely on on two-phase commit and read-write locks for concurrency control. They use a global naming service to locate objects and persist them on stable storage. Orca [7] is a programming language with support for distributed objects. It is part of the Amoeba operating system [58]. In all of the above systems, arguments of a remote call are solely passed by values. As explained in Section 3.4, AOF supports in addition references to other distributed objects.

Some works move from a remotely accessible object model to a wrapper encompassing both the proxies and the object's replicas. Globe [59] is a wide-area distributed system built as a middleware of objects. An object defines its own policies and mechanisms for replication, migration, communication and concurrency control. Each such concern is implemented as a sub-object using reflective object-oriented programming [39]. A fragmented object [55] consists of multiple fragments located on multiple machines. The fragmented object model extends the traditional concept of stub-based distributed objects. It allows to arbitrary partition the state and functionalities of an object across its fragments. Core concerns such as run-time protection, migration and the auto-generation of stubs are mentioned. In comparison to AOF, such approaches require experienced programmers to fully leverage their capabilities.

The object group design pattern. Java Remote Method Invocation (RMI) and CORBA [43] are two proeminent im-



(a) PostgreSQL *(2 proxies, stored procedures on, persistence on)*

(b) AOF *(one replica per object, persistence on)*

(c) Ubuntu One trace: `doCommit` operation latency

Figure 6: Comparing AOF with PostgreSQL with the StackSync application

plementations of the distributed object paradigm. In the context of CORBA, systems such as Electra [47], Eternal [51] and GARF [25, 31] provide fault-tolerant distributed objects using the *object group* design pattern [48]. Under this pattern, the Object Request Broker (ORB) delegates each method call to an appropriate group of processes that answer it. Inside a process group, communication takes place using dedicated dependable primitives. The ORB also relies on some form of state-machine replication to deal with problems related to call idempotence at the client and the server (named response collation and request filtering respectively in [25]). We explain how AOF addresses such issues in Section 4.

To coordinate the distributed copies of a logically shared object, Isis [10] introduces the view synchrony paradigm, while Jgroup [50] employs a partitionable group communication service. In this latter solution, a partitioned set of processes may continue updating its copy of the distributed object, but as expected by the CAP impossibility result [27], without providing guarantees on consistency. The authors of [24] give a critical look at the object group design pattern. They identify the group proliferation as a core problem. We avoid this problem in AOF using a higher-level abstraction, the listenable key-value store.

Object-oriented databases. Persistence is an orthogonal concern of programming languages [4], and a key motivation behind the object-oriented databases movement [5, 35]. Object-oriented databases provide support for persistent and nested objects and do not require a mapping phase to an intermediate representation. O++ [1] extends C++ with (limited) querying capabilities and adds object persistence via the `pnew` keyword. AOF offers a simple language-level support of per object and per type data persistence (see Section 2.1). The correspondence between an in-memory object and its representation to stable storage relies on the Java support and is consequently type safe.

DB4O is a discontinued object-oriented database. Unlike AOF, it does not support remote method evaluation close to the data; instead it requires fetching the object locally, and re-uploading it after mutation. There is no guarantee for consistency when two processes fetch and update the same object. Versant is a closed-source follow-up to DB4O that supports replication. After each mutation, the system requires to ship the full state of the object to the replicas, instead of coordinating operations as with AOF's state-machine replication approach. Other examples of object-oriented databases are found in the context of embedded systems, such as Perst and Realm. These databases support multi-threaded access and ACID semantics for transactions on multiple objects, but target a different deployment context than AOF, with no necessity for replication.

Object-relational databases. Object-relational databases are similar to relational ones with object-oriented specificities. One prominent system in this category is Postgres [57]. Postgres supports user-specified types, stored procedures and type inheritance. Many of the ideas coming from object-relational databases have become incorporated into the SQL:1999 standard via structured types, and are now available in commercial databases. The SQL:1999 standard also introduces language constructs to overcome the inherent limitations of relational algebra, such as the absence of a transitive closure operator [3].

Kossmann et al. [41] study alternatives to databases for data management in the Cloud. They conclude that a Distributed Control Architecture is potentially the best match for reaching scalability and elasticity. A Distributed Control Architecture is a shared-disk architecture with loose coupling between nodes. The authors present an example of this architecture [12] that relies on a scalable and elastic put/get storage layer (Amazon S3). AOF can be considered a Distributed Control Architecture where, instead of building a relational database on top of a NoSQL one, we provide a programming framework for distributed objects. Similarly to [12], AOF benefits from the scalability and elasticity capabilities of the underlying NoSQL storage. In addition, it offers a simple interface to transparently build distributed object-oriented applications.

Dedicated data structures. Over the past few years, multiple dedicated NoSQL solutions focusing on a particular data type have appeared, e.g., trees [34], locks [13], tuple spaces [9, 26] and sets [14]. Applications generally use them to manipulate metadata and for coordination purposes. Kalantari and Schiper [37] show that such approaches have a limited versatility. The Extensible Coordination approach [21] addresses this issue, allowing clients to embed user-defined operations directly in the storage layer. These operations should however manipulate only the ZooKeeper tree [34]. With AOF, the developer is not bound to a particular interface. Moreover, our framework allows non-blocking interprocess synchronization using distributed objects, and blocking synchronization with polling.

8. CONCLUSION

This paper presents AOF, a novel implementation of the distributed object paradigm. Our framework is built upon a novel NoSQL database, the Listenable Key Value Store (LKVS), that we introduce in detail. AOF allows distributed applications to take advantage of this dependable, scalable and elastic back end for their objects, without the need for a complex translation layer. It provides a simple and transparent interface to the programmer who only has to declare which of her (composed) objects and fields must be hosted reliably and persistently by the NoSQL back end. Operations on these objects are evaluated directly at the LKVS servers, using an extension of state-machine replication.

Our paper also offers an in-depth comparison between AOF and the object-relational mapping (ORM) approach. This comparison relies on the use of state-of-the-art techniques for relational database, such as sharding and stored procedures. While AOF is leaner and simpler, it allows features that a sharded database does not provide. In particular, AOF can support strong consistency over all application objects, and offers elastic scalability.

To assess in practice this comparison, we detail two implementations of a metadata back end for a personal cloud storage service. Our evaluation shows that AOF offers significant benefits without performance penalty. Furthermore, when the application is sufficiently parallel, AOF outperforms the ORM-based implementation both in throughput and latency. In particular, we show that this result holds for a trace obtained with the real-world deployment of a personal cloud storage service (Ubuntu One).

An interesting follow-up of this work is the addition of querying capabilities to the distributed objects, to avoid the maintenance of explicit indexes in the common case. We envision this extension as our immediate future work, targeting an integration with technologies such as Hibernate Search.

Acknowledgments

We are grateful to Valerio Schiavoni for comments on drafts of this work, and to Marc Shapiro for fruitful discussions on the topic. The work of P. Sutra, E. Rivière, E. Bernard, W. Burns and G. Zamarreño was supported by the LEADS project funded by the European Commission under the Seventh Framework Program (grant 318809). This work of C. Cotes, M. Artigas and P. Lopez has been partly funded by the EU project H2020 "IOStack: Software-Defined Storage for Big Data" (644182) and Spanish research project "Cloud Services and Community Clouds" (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] R. Agrawal and N. H. Gehani. Ode (object database and environment): The language and the data model. *SIGMOD Rec.*, 18(2):36–45, June 1989. ISSN 0163-5808.

[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102, 2010.

[3] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *POPL*, 1979.

[4] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. Readings in object-oriented database systems. chapter An Approach to Persistent Programming, pages 141–146. 1990. ISBN 0-55860-000-0.

[5] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, and Stanley Zdonik. Building an object-oriented database system. chapter The Object-oriented Database System Manifesto. Morgan Kaufmann, 1992.

[6] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8): 55–59, August 1977. ISSN 0362-1340.

[7] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3): 190–205, March 1992. ISSN 0098-5589.

[8] Christian Bauer, Gavin King, and Gary Gregory. *Java Persistence with Hibernate*. Manning Publications, Greenwich, CT, USA, 2015. ISBN 1617290459.

[9] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Eurosys*, 2008.

[10] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21 (5):123–138, November 1987. ISSN 0163-5980.

[11] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2 (1):39–59, February 1984. ISSN 0734-2071.

[12] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD*, 2008.

[13] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[14] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013. ISBN 1617290858, 9781617290855.

[15] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071.

[17] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4), 2001.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[21] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *EuroSys*, 2015.

[22] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *IMC*, 2013.

[23] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *SIGCOMM*, 2012.

[24] Pascal Felber and Priya Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Trans. Computers*, 53(5):497–511, 2004.

[25] Pascal Felber, Rachid Guerraoui, and André Schiper. The implementation of a CORBA group communication service. *Theory and Practice of Object Systems*, 4(2): 93–105, 1998.

[26] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. ISSN 0164-0925.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[28] R. Gracia-Tinedo, M. Sanchez Artigas, A. Moreno-Martinez, C. Cotes, and P. Garcia Lopez. Actively measuring personal cloud storage. In *IEEE Cloud*, 2013.

[29] Raul Gracia Tinedo, Tian Yongchao, Josep Sampe, Harkous Hamza, John Lenton, Pedro Garcia Lopez, Marc Sanchez Artigas, and Marko Vukolic. Dissecting ubuntuOne: Autopsy of a global-scale personal Cloud back-end. In *IMC 2015, ACM Internet Measurement Conference, October 28-30, 2015, Tokyo, Japan*, Tokyo, JAPON, 10 2015.

[30] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2), 2001.

[31] Rachid Guerraoui, Pascal Felber, Benoît Garbinato, and Karim Mazouni. System support for object groups. In *OOPSLA*, 1998.

[32] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. ISSN 0164-0925.

[33] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys.*, 12(3), 1990.

[34] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[35] H. Ishikawa. *Object-Oriented Database System*. Springer-Verlag, 1993.

[36] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, 1994.

[37] Babak Kalantari and André Schiper. Addressing the zookeeper synchronization inefficiency. In *ICDCN*, 2013.

[38] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

[39] G. Kiczales, J.D. Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[40] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[41] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, 2010.

[42] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979. ISSN 0018-9340.

[43] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1), 1997. ISSN 1096-9942.

[44] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, March 1988. ISSN 0001-0782.

[45] Songbin Liu, Xiaomeng Huang, Haohuan Fu, and Guangwen Yang. Understanding data characteristics and access patterns in a cloud storage system. In *CCGrid*, 2013.

[46] Pedro Garcia Lopez, Marc Sanchez-Artigas, Sergi Toda, Cristian Cotes, and John Lenton. Stacksync: Bringing elasticity to dropbox-like file synchronization. In *Middleware*, 2014.

[47] Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *COOTS*, 1995.

[48] Silvano Maffeis. The object group design pattern. In *COOTS*, 1996.

[49] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[50] Alberto Montresor. The Jgroup reliable distributed object model. In *DAIS*, 1999.

[51] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Consistency of partitionable object groups in a CORBA framework. In *ICSS*, 1997.

[52] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The design and implementation of arjuna. *Computing Systems*, 8(2): 255–308, 1995.

[53] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. ISSN 0360-0300.

[54] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *IEEE SRDS*, 1986.

[55] Marc Shapiro, Yvon Gourbant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: an object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[56] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–564, October 1990.

[57] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, October 1991. ISSN 0001-0782.

[58] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert van Renesse, and Henri E. Bal. The amoeba distributed operating system - A status report. *Computer Communications*, 14(6):324–335, 1991.

[59] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: a wide area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

[60] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009. ISSN 0001-0782.

APPENDIX

A. AN ELASTIC LKVS

In this appendix, we describe an algorithmic construction of a Listenable Key-Value Store (LKVS). Our implementation of the LKVS abstraction in Infinispan (see Section 4) follows this construction. As detailed below, it offers the interface and guarantees we specified in Section 3.2.

Before delving into further details, we should first observe that building a regular key-value store, or equivalently a distributed shared map, boils down to implementing a set of atomic registers. In the case where the map is elastic [2], this is achievable in a fully asynchronous distributed system. When the map is listenable, that is a LKVS abstraction, this is more intricate.

Indeed, we can easily show that consensus is implementable with a wait-free and linearizable LKVS. As a direct consequence of Herlihy's hierarchy [32], we should thus base any implementation of the LKVS interface on top of a powerful group communication primitive. In our case, we consider a variation of the view synchrony [17] paradigm. We introduce this variation below, then cover an elastic and disjoint-access parallel LKVS implementation.

System Model. In what follows, we consider the usual model of a distributed system. More precisely, we shall assume a set of distributed processes Π forming a fully-connected network. Each process may fail, and upon a failure interrupt its execution of the program – this is the well-know fail-stop or *crash* model. The link between any two processes is reliable. This means that if a process p sends a message m to some correct process q, eventually q receives m. We make no synchrony assumption for the moment, and shall detail them where appropriate.

View Synchrony. Let us name any non-empty subset V of Π a *view*. View synchrony allows a process to send (*send*) and receive (*receive*) messages, as well as to get notified upon a view change (*viewChange*). When the event *viewChange*(V) occurs at some process p, we say that view V is installed at p; initially all the processes install some view $V_0 \subseteq \Pi$.

Successive views installations during a run guarantee that (Primary Component) they follow some total order; (View Synchrony) processes agree on the messages they receive in a view ; as well as (Total Order) they also agree on the order in which they receive such messages. Furthermore, we shall be considering that the (Precise Membership) property holds, i.e., *(i)* every faulty process gets eventually excluded from the view, and *(ii)* every correct process is eventually present in the view.

In the original paradigm [17], messages are addressed to all processes in the view. Differently, we consider in our case that a message can be addressed to a subset of V. We note that such a variation is implementable using the original view synchrony primitive, discarding appropriately the messages. A more efficient implementation can ensure disjoint-access parallelism. This is achievable for instance with the help of genuine atomic multicast [30] and an eventually perfect failure detector [15]. Details are however outside of the range of this paper.

Replication. Based on the notion of view synchrony, we define a replication mapping as follows. Let us consider some replication factor ρ. For some view V and some key k, we assume a locally computable function $rep(k)_V$ that returns the ρ processes in charge of replicating the item under key k in view V. A replication mapping can be implemented, for instance, with the consistent hashing strategy [38].

Algorithm. We detail our elastic disjoint-access parallel implementation of the LKVS abstraction in Algorithm 2. This algorithm executes at each process a client and a server task. The client task models calls at the LKVS interface. The server task contains the core of our construction.

Let us note $\mathbb{K}$ the set of keys, and $\mathbb{U}$ the values they can take. As detailed next, Algorithm 2 makes use of three variables:
- $\mathcal{V}$ stores the view;
- For some key k, $\mathcal{D}(k)$ indicates the value of the local copy of k (in $\mathbb{U} \cup \{\perp\}$); and
- $\mathcal{F}(k)$ holds the set of filter-containers registered at key k.

Initially at every process, $\mathcal{D}(k)$ equals $\perp$, i.e., the local value is null, and $\mathcal{F}(k)$ equals $\varnothing$.

When the client task executes an operation accessing a key k, it sends an appropriate message to the set of replicas of key k in the current view ($rep(k)_{\mathcal{V}}$). Upon receiving such a message, if the client asks for the current value of k (line 21), or wishes to register or unregister a listener (lines 32 and 36),

the replica executes the corresponding action. If, on the other hand, the client updates the shared map with a value (k, u), a replica of k first notifies the filter-converters registered at key k (lines 26 to 29), then it updates $\mathcal{D}(k)$ appropriately (line 30), before returning an acknowledgment to the client.

Upon receiving an event $viewChange(V)$, process p computes the set of keys for which it is newly in charge (line 42). For every key k that it replicates, process p also computes all the pairs (k, q) for which process q is a new replica in view V (line 43). For every such pair (k, q), p sends to process q the state of k together with the filter-converters registered in the mapping $\mathcal{F}(k)$. Then, process p waits that until such information is received for every new key that it replicates (line 45).

Correctness. Algorithm 2 implements a wait-free linearizable LKVS abstraction. Before proving such a claim, we introduce a few assumptions that were omitted from Algorithm 2 for clarity. First, (A1) a message sent in a view V is not delivered in a view $V' \neq V$. Second, (A2) both the client and server tasks resend a message if a view change occurs and that message was not delivered in the previous view. Finally, we consider that (A3) when view V' follows view V, for any key k, there exists a process in $rep(k)_V \cap V'$, and all the processes in V' have the time to execute the view change (lines 41 to 47).

THEOREM 1. *Algorithm 2 implements a wait-free linearizable* LKVS.

PROOF. First of all, we prove that for each key k, the operations accessing k are linearizable. From the locality property of linearizability [33], this result implies that the full LKVS interface is linearizable.

Consider some run ρ of Algorithm 2. Name $h = (E_h, <_h)$ the projection of the history produced by ρ over the client operations accessing key k.

As a starter, assume that all processes stay in the initial view V_0 during run ρ. We observe that they all hold the same values for variables $\mathcal{D}(k)$ and $\mathcal{F}(k)$ at the beginning of ρ. Then, by the total order property of view synchrony, they receive `PUT`, `GET`, `REG` and `UREG` messages in the same order. Hence, from the pseudo-code of Algorithm 2, we easily deduce that $(E_h, <_k)$ is a linearization of h.

Now, consider that view changes occur during ρ. From the definition of view synchrony, these view changes take place in some total order. Let $V_0, V_1, \ldots$ be such a sequence, and consider some view $V_i \geq 0$. We proceed by induction, considering that *(i)* history h is linearized up to view V_i, and *(ii)* there exist values v and F such that before applying the view change at line 47, each process in $rep(k)_{V_i}$ is either crashed or alive with its variables $\mathcal{D}(k)$ and $\mathcal{F}(k)$ equal to respectively v and F.

Recall that by assumption A3 at least one replica belongs in $rep(k)_{V_i} \cap V_{i+1}$, and that all the processes in V_{i+1} have the time to execute the view change. If $rep(k)_{V_i} = rep(k)_{V_{i+1}}$, the code at line 42 tells us that variable I does not contain k. Hence, neither $\mathcal{D}(k)$ nor $\mathcal{F}(k)$ change in V_{i+1}. Otherwise, if q belongs to $V_{i+1} \setminus V_i$, we observe from assumption A3 and our induction hypothesis that after executing lines 42 to 46, process q necessarily holds $\mathcal{D}(k) = v$ and $\mathcal{F}(k) = F$. Then, since assumption A1 holds, processes in $rep(k)_{V_{i+1}}$ agree on the sequence of operations on key k they receive during view V_{i+1}. As a consequence, we may reduce this case to the situation where no view change occurs, implying that linearizability holds in h up to view V_{i+1} included.

We now turn our attention to the liveness property of Algorithm 2. Let us consider that the client task executes at some correct process, invoking an operation op of the LKVS interface in some view V. Let t be the first time at which all faulty processes have crashed. From the precise membership property of view synchrony, there exists a time $t' > t$ from which the view does not change and where all correct processes are included in the view. We name V_s this stable view built (possibly) after time t' and consider the following two cases:

($V = V_s$) Since *(i)* links are reliable, and *(ii)* for every key k, the replica set of k is non-empty in V_s, according to the pseudo-code of Algorithm 2, operation op eventually returns in V_s.

($V \neq V_s$) According to the pseudo-code of Algorithm 2, op returns in view V, or there exists a message m in its critical path such that m is not delivered in view V. In such a case, assumption A2 tells us that the sending process re-sends message m in the next installed view. Hence, in the worst case, message m is eventually re-sent in view V_s, and this

Algorithm 2 LKVS– code at process p

```
1:  Variables:
2:      D                                   // Initially, ∀k ∈ 𝕂 : D(k) = ⊥
3:      F                                   // Initially, ∀k ∈ 𝕂 : F(k) = ∅
4:      V                                   // Initially, V_0
5:
6:  Task Client
7:      get(k) :=
8:          eff: send ⟨GET, k⟩ to rep(k)_V
9:               wait until received ⟨ACK, u⟩
10:              return u
11:     put(k, u) :=
12:         eff: send ⟨PUT, k, u⟩ to rep(k)_V
13:              wait until received ⟨ACK⟩
14:     regListener(k, l, f_l) :=
15:         eff: send ⟨REG, k, l, f_l⟩ to rep(k)_V
16:              wait until received ⟨ACK⟩
17:     unregListener(k, l) :=
18:         eff: send ⟨UREG, k, l⟩ to rep(k)_V
19:              wait until received ⟨ACK⟩
20: Task Server
21:     doGet(k) :=
22:         pre: received ⟨GET, k⟩ from q
23:         eff: send ⟨ACK, D(k)⟩ to q
24:     doPut(k, u) :=
25:         pre: received ⟨PUT, k, u⟩ from q
26:         eff: forall f_l ∈ F(k) do
27:                r ← f_l.filter(u, D(k))
28:                if r ≠ ⊥
29:                  send ⟨k, r⟩ to l
30:              D(k) ← u
31:              send ⟨ACK⟩ to q
32:     doRegListener(k, l, f_l) :=
33:         pre: received ⟨REG, k, l, f_l⟩ from q
34:         eff: F(k) ← F(k) ∪ {f_l}
35:              send ⟨ACK⟩ to q
36:     doUnregListener(k, l) :=
37:         pre: received ⟨UREG, k, l⟩ from q
38:         eff: F(k) ← F(k) \ {f_l}
39:              send ⟨ACK⟩ to q
40:     doViewChange :=
41:         pre: viewChange(V)
42:         eff: I ← {k : p ∈ rep(k)_V \ rep(k)_V}
43:              O ← {(k, q) : p ∈ rep(k)_V, q ∈ rep(k)_V \ rep(k)_V}
44:              forall (k, q) ∈ O do send ⟨k, D(k), F(k)⟩ to q
45:              wait until ∀k ∈ I : received ⟨k, d_k, F_k⟩
46:              forall k ∈ I do (D(k), F(k)) ← (d_k, F_k)
47:              V ← V
```

case boils down to the previous one.

This leads us to conclude that client operations are wait-free. □

It remains to prove that Algorithm 2 is disjoint-access parallel and elastic.

Elasticity states that the number of distributed processes in use by the implementation of the LKVS can shrink/grow without interruption. This property follows immediately from the view synchrony paradigm at the core of Algorithm 2.

On the other hand, disjoint-access parallelism [36] requires that operations on distinct keys access distinct base objects. In our setting, this means that these operations either access distinct replicas, or at a replica, they access distinct memory locations. According to its pseudo-code, Algorithm 2 ensures this guarantee in the common case, i.e., when the system is synchronous, failure-free and when no view change occurs.

Before we close this section, let us note that Algorithm 2 splits clients and servers in two tasks operating on the same set of machines. Our implementation (see Section 4) does not have this limitation. Clients actually operate in the interface tier, sending calls to the LKVS machines that themselves run Algorithm 2. This raises the question of call idempotence. As explained in Section 3.4, AOF solves this problem in the object management tier.

B. CORRECTNESS OF AOF

Section 2.2 lists several guarantees AOF offers to the programmer of a distributed application. The previous section shows that the LKVS and thus AOF ensures elasticity and disjoint-access parallelism at the storage tier. It thus remains to prove that method calls to distributed objects are wait-free, and that these objects are also both linearizable and persistent. In what follows, we state rigorously that AOF satisfies these three guarantees.

Liveness. First of all, we establish that the interface of AOF is wait-free. To this end, consider that an application process p invokes $newObject(k, T, args)$ to create some distributed object o. According to Section 3.3, AOF immediately returns to such a call a proxy object stored in a local container c. Next, we prove that every method call to object o via this proxy eventually returns.

PROPOSITION 1. *It is always true that either* (i) *o never existed (the value stored at key ref_o in the* LKVS *equals $\bot$),* (ii) *object o exists at its object component ($s \neq \bot$ in Algorithm 1), or* (iii) *no container referring to o is opened and o can be fetched from persistent storage (the value stored at key ref_o equals $\langle \texttt{PER}, \_, \_ \rangle$).*

PROOF. We proceed by induction, considering all the insertions at key ref_o. Initially, as a call to $get(ref_o)$ returns $\bot$, the invariant holds.

Now, consider that an insertion occurs at key ref_o on the LKVS. This triggers the execution of $filter(u, v)$ at the object component.

Consider that $s = \bot$ holds at the object component. According to Section 3, solely a container c referring to object o may execute an insertion at key ref_o. We then consider two cases.

- If the insertion corresponds to the opening of c, then the object component executes $filter(u, v)$ with $u = \langle \texttt{OPEN}, c, \_ \rangle$. From the code between line 11 and line 16 in Algorithm 1 and the fact that the invariant holds, we deduce that $s \neq \bot$ occurs when $filter(u, v)$ returns.
- Now, assume that this insertion is either the closing of c or the invocation of some method. In this case, c is already opened. Name t and t' the times at which the object component executes respectively $filter(\langle \texttt{OPEN}, c, \_ \rangle, \_)$ and $s \leftarrow \bot$ at line 31. If $t < t'$, then the invariant holds at time t and the object component must execute line 10. Thus, $|openContainers| > 0$ holds at time t' and the test at line 30 fails. Otherwise $t > t'$ and in such case $s \neq \bot$ holds after $filter(\langle \texttt{OPEN}, c, \_ \rangle, \_)$ returns. Thus, we obtain a contradiction in both cases.

Then, assume that $s \neq \bot$ is true at the object component. The invariant holds after $filter(u, v)$ returns, unless the object component assigns value $\bot$ to s at line 31. In such a case, the precondition at line 29 implies that the last value stored under key ref_o equals $\langle \texttt{PER}, \_, \_ \rangle$.

Then, consider for the sake of contradiction that a container c is still opened. When the object component executed $filter(\langle \texttt{OPEN}, c, \_ \rangle, \_)$, the invariant was holding. This leads to $|openContainers| > 0$, contradicting that the test at line 30 succeeds. □

From Proposition 1, we immediately deduce the following result.

COROLLARY 1. *If some container c is opened then $s \neq \bot$ holds at the object component of o.*

In the theorem below, we state the core liveness property of the distributed objects implemented with the help of AOF.

THEOREM 2. *All method calls to a distributed object o are wait-free.*

PROOF. AOF ensures that container c is opened before invoking method m. This means that AOF first registers container c and its associated filter-converter f_c with a call $regListener(ref_o, c, f_c)$ to the LKVS. Then, AOF executes $put(ref_o, \langle \texttt{OPEN}, c \rangle)$ to actually open c. As the LKVS interface is wait-free, these two calls return.

Assume from now on that container c is opened. AOF registers a future for the call to method m. Then, it executes $put(ref_o, \langle \texttt{INV}, c, m \rangle)$ at the LKVS interface. Since calls at this interface are wait-free, the filter-converter f_c is eventually notified with $filter(\langle \texttt{INV}, c, m \rangle, \_)$.

The composite component first calls the identity component. Since $\langle \texttt{INV}, c, m \rangle$ contains the identity of container c, the identity component returns a non-null value. Hence, the composite component proceeds to calling the object component.

From Theorem 1, as container c is opened, the precondition at line 19 holds. Thus, the object component returns $\langle \text{ACK}, r \rangle$, for some response r computed at line 20 in Algorithm 1. This pair is also the result of the call $filter(\langle \texttt{INV}, c, m \rangle, \_)$ at the composite component. As $\langle \text{ACK}, r \rangle \neq \bot$, the listener (i.e., container c) eventually receives this value from the filter-converter f_c. Finally, value r is returned to the calling process by fulfilling the appropriate future. □

Safety. To start, let us consider as above that some application process p invokes $newObject(k, T, args)$ to create a

distributed object o. AOF returns to such a call a proxy stored in a local container c. We note here that this proxy is of the correct type T, and that it implements object o at process p.

In Section 3, we pointed out that application process p is not the only one to manipulate object o via its container and that concurrency might occur. Moreover, we shall also consider that the full application might stop then later restart. We model these two constraints with the assumption that an unbounded amount of process may manipulate object o. Our next result shows that in such case all the methods calls to object o are linearizable.

THEOREM 3. *Object o is linearizable.*

PROOF. Consider a run ρ of AOF. Let h be the history obtained from ρ by only considering method calls that affect object o. Since we prove in Theorem 2 that such calls are wait-free, we may consider that h is complete, i.e., every invocation in h has a matching response in h.

We use $.st$ and $.val$ selectors to respectively extract the state and the response value components of a method call, i.e., given some state s and some method m, $\tau(s,m) = (\tau(s,m).st, \tau(s,m).val)$.

Consider a response value r to some method m by an application process in history h. This corresponds to the value $\tau(s,m).val$, where s is the state of object o at the object component when it executes line 21 in Algorithm 1. Thus, the object component previously executed $\mathit{filter}(u, \_)$ with $u = \langle \texttt{INV}, c, m\rangle$. By the fact that the underlying LKVS is linearizable, there exists some order $\ll$ on the set of such events that occurred during ρ. We name l the linearization of the operations in h according to $\ll$. In what follows, we prove that l is a linearization of h.

As the LKVS is linearizable, calls to $\mathit{filter}()$ occur in the order the linearization of the corresponding $\mathit{put}()$ operations take place. Hence, it is clear that if m precedes m' in h then it is also the case in l. It remains to be shown that l satisfies the sequential specification of o. We proceed by induction on the events occurring in l. We name s_0 the initial value of variable s at the object component after the execution of line 12.

Let us define function τ^+ by repeated application of τ, i.e., given a sequence of methods $\sigma = \langle m_1, \ldots, m_{n\geq 1}\rangle$ and a state s:

$$\tau^+(s,\sigma) \triangleq \begin{cases} \tau(s, m_1) & \text{if } n = 1, \\ \tau^+(\tau(st, m_1).st, \langle m_2, \ldots, m_n\rangle) & \text{otherwise.} \end{cases}$$

Now consider a method m associated with some event in l. Note $\sigma = \langle m_1, \ldots, m_{n\geq 1}\rangle$ the methods preceding m in l.

We claim that when the object component applies method m to variable s at line 20, it has applied previously all the sequence σ to its local variable s. In formal terms, this means that before applying m at line 20 we have:

$$s = \tau^+(s_0, \sigma).st \tag{1}$$

If m is the smallest element in l, the above result is straightforward. Otherwise, we consider the events corresponding to response of m_n and the invocation of m in ρ, i.e., respectively the response to $\mathit{filter}(\langle \texttt{INV}, \_, m_n\rangle)$ and the invocation of $\mathit{filter}(\langle \texttt{INV}, \_, m\rangle)$.

By induction hypothesis, variable s equals $\tau^+(s_0,\sigma).st$ right after the response to $\mathit{filter}(\langle \texttt{INV}, \_, m_n\rangle)$. Hereafter, we show that this is also the case just before the invocation of $\mathit{filter}(\langle \texttt{INV}, \_, m\rangle)$. To this end, we consider whether the object component executes line 31 or not between these two events.

- If this is not the case, since m immediately follows m_n in the order $\ll$, variable s does not change. Thus, equation (1) is true.

- Otherwise, let t be the time at which the object component executes line 31. Note c the container that invokes m. From Proposition 1 and our induction hypothesis, the LKVS holds at key ref_o the value s and c is closed at time t. Hence, c is opened between time t and the invocation of $\mathit{filter}(\langle \texttt{INV}, \_, m\rangle)$.

  Without lack of generality, assume that during this period of time c is the first container opened and it is never closed. At the object component, this corresponds to a call $\mathit{filter}(\_, \langle \texttt{PER}, c, \tau^+(s_0,\sigma).st\rangle)$. Since $s = \bot$ holds, the object component assign $\tau^+(s_0,\sigma).st$ to s. Because c remains open and m immediately follows m_n in the order $\ll$, variable s never changes hereafter.

We note r the response value of m in l, that is $r = \tau(s,m).val$. From (1), it follows that $r = \tau(\tau^+(s_0,\sigma), m).val$, and history $l_{|\leq m}$ satisfies the sequential specification of o. □

From the locality property of linearizability, and the fact that AOF ensures call idempotence (see Section 4 for the implementation details), Theorem 3 implies that composing distributed objects with AOF maintains linearizability.

Persistence. In the theorem below, we prove that the guarantee of persistence is true for the distributed objects created with AOF.

THEOREM 4. *Object o is persistent.*

PROOF. At some point in time, there might exist at most one object component for object o. If this component is destroyed then no composite component points to it and no filter-converter containing the object component for o might exist as well. As a consequence, all containers referring to object o are closed. Proposition 1 tells us that in such case o can be fetched from persistent storage. This shows that if object o exists at some point in time, then it exists forever. Hence, object o is persistent. □